I 47

47.1 IDIV—Signed Divide

Opcode	Instruction	Description
F6 /7	IDIV *r/m8*	Signed divide AX (where AH must contain sign-extension of AL) by *r/m* byte. (Results: AL=Quotient, AH=Remainder)
F7 /7	IDIV *r/m16*	Signed divide DX:AX (where DX must contain sign-extension of AX) by *r/m* word. (Results: AX=Quotient, DX=Remainder)
F7 /7	IDIV *r/m32*	Signed divide EDX:EAX (where EDX must contain sign-extension of EAX) by *r/m* doubleword. (Results: EAX=Quotient, EDX=Remainder)

Description

Divides (signed) the value in the AL, AX, or EAX register by the source operand and stores the result in the AX, DX:AX, or EDX:EAX registers. The source operand can be a general-purpose register or a memory location. The action of this instruction depends on the operand size, as shown in the following table:

Operand Size	Dividend	Divisor	Quotient	Remainder	Quotient Range
Word/byte	AX	r/m8	AL	AH	–128 to +127
Doubleword/word	DX:AX	r/m16	AX	DX	–32,768 to +32,767
Quadword/doubleword	EDX:EAX	r/m32	EAX	EDX	-2^{31} to $2^{32} - 1$

Non-integral results are truncated (chopped) towards 0. The sign of the remainder is always the same as the sign of the dividend. The absolute value of the remainder is always less than the absolute value of the divisor. Overflow is indicated with the #DE (divide error) exception rather than with the OF (overflow) flag.

Operation

```
IF SRC = 0
    THEN #DE; (* divide error *)
FI;
IF OpernadSize = 8 (* word/byte operation *)
    THEN
        temp ← AX / SRC; (* signed division *)
        IF (temp > 7FH) OR (temp < 80H)
        (* if a positive result is greater than 7FH or a negative result is less than 80H *)
            THEN #DE; (* divide error *) ;
            ELSE
                AL ← temp;
                AH ← AX SignedModulus SRC;
        FI;
    ELSE
        IF OpernadSize = 16 (* doubleword/word operation *)
            THEN
                temp ← DX:AX / SRC; (* signed division *)
                IF (temp > 7FFFH) OR (temp < 8000H)
                (* if a positive result is greater than 7FFFH *)
                (* or a negative result is less than 8000H *)
                    THEN #DE; (* divide error *) ;
                    ELSE
```

```
                        AX ← temp;
                        DX ← DX:AX SignedModulus SRC;
                    FI;
                ELSE (* quadword/doubleword operation *)
                    temp ← EDX:EAX / SRC; (* signed division *)
                    IF (temp > 7FFFFFFFH) OR (temp < 80000000H)
                    (* if a positive result is greater than 7FFFFFFFH *)
                    (* or a negative result is less than 80000000H *)
                        THEN #DE; (* divide error *) ;
                        ELSE
                            EAX ← temp;
                            EDX ← EDXE:AX SignedModulus SRC;
                    FI;
        FI;
FI;
```

Flags Affected

The CF, OF, SF, ZF, AF, and PF flags are undefined.

Protected Mode Exceptions

#DE	If the source operand (divisor) is 0.
	The signed result (quotient) is too large for the destination.
#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#DE	If the source operand (divisor) is 0.
	The signed result (quotient) is too large for the destination.
#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.

Virtual-8086 Mode Exceptions

#DE	If the source operand (divisor) is 0.
	The signed result (quotient) is too large for the destination.
#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

47.2 IMUL—Signed Multiply

Opcode	Instruction	Description
F6 /5	IMUL *r/m8*	AX← AL ∗ *r/m* byte
F7 /5	IMUL *r/m16*	DX:AX ← AX ∗ *r/m* word
F7 /5	IMUL *r/m32*	EDX:EAX ← EAX ∗ *r/m* doubleword
0F AF /*r*	IMUL *r16,r/m16*	word register ← word register ∗ *r/m* word
0F AF /*r*	IMUL *r32,r/m32*	doubleword register ← doubleword register ∗ *r/m* doubleword
6B /*r ib*	IMUL *r16,r/m16,imm8*	word register ← *r/m16* ∗ sign-extended immediate byte
6B /*r ib*	IMUL *r32,r/m32,imm8*	doubleword register ← *r/m32* ∗ sign-extended immediate byte
6B /*r ib*	IMUL *r16,imm8*	word register ← word register ∗ sign-extended immediate byte
6B /*r ib*	IMUL *r32,imm8*	doubleword register ← doubleword register ∗ sign-extended immediate byte
69 /*r iw*	IMUL *r16,r/m16,imm16*	word register ← *r/m16* ∗ immediate word
69 /*r id*	IMUL *r32,r/m32,imm32*	doubleword register ← *r/m32* ∗ immediate doubleword
69 /*r iw*	IMUL *r16,imm16*	word register ← *r/m16* ∗ immediate word
69 /*r id*	IMUL *r32,imm32*	doubleword register ← *r/m32* ∗ immediate doubleword

Description

Performs a signed multiplication of two operands. This instruction has three forms, depending on the number of operands.

- **One-operand form.** This form is identical to that used by the MUL instruction. Here, the source operand (in a general-purpose register or memory location) is multiplied by the value in the AL, AX, or EAX register (depending on the operand size) and the product is stored in the AX, DX:AX, or EDX:EAX registers, respectively.
- **Two-operand form.** With this form the destination operand (the first operand) is multiplied by the source operand (second operand). The destination operand is a general-purpose register and the source operand is an immediate value, a general-purpose register, or a memory location. The product is then stored in the destination operand location.
- **Three-operand form.** This form requires a destination operand (the first operand) and two source operands (the second and the third operands). Here, the first source operand (which can be a general-purpose register or a memory location) is multiplied by the second source operand (an immediate value). The product is then stored in the destination operand (a general-purpose register).

When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.

The CF and OF flags are set when significant bits are carried into the upper half of the result. The CF and OF flags are cleared when the result fits exactly in the lower half of the result.

The three forms of the IMUL instruction are similar in that the length of the product is calculated to twice the length of the operands. With the one-operand form, the product is stored exactly in the destination. With the two- and three- operand forms, however, result is truncated to the length of the destination before it is stored in the destination register. Because of this truncation, the CF or OF flag should be tested to ensure that no significant bits are lost.

The two- and three-operand forms may also be used with unsigned operands because the lower half of the product is the same regardless if the operands are signed or unsigned. The CF and OF flags, however, cannot be used to determine if the upper half of the result is non-zero.

Operation

```
IF (NumberOfOperands = 1)
    THEN IF (OperandSize = 8)
        THEN
            AX ← AL ∗ SRC  (* signed multiplication *)
            IF ((AH = 00H) OR (AH = FFH))
                THEN CF = 0; OF = 0;
                ELSE CF = 1; OF = 1;
            FI;
        ELSE IF OperandSize = 16
            THEN
                DX:AX ← AX ∗ SRC  (* signed multiplication *)
                IF ((DX = 0000H) OR (DX = FFFFH))
                    THEN CF = 0; OF = 0;
                    ELSE CF = 1; OF = 1;
                FI;
            ELSE (* OperandSize = 32 *)
                EDX:EAX ← EAX ∗ SRC  (* signed multiplication *)
                IF ((EDX = 00000000H) OR (EDX = FFFFFFFFH))
                    THEN CF = 0; OF = 0;
                    ELSE CF = 1; OF = 1;
                FI;
        FI;
    ELSE IF (NumberOfOperands = 2)
        THEN
            temp ← DEST ∗ SRC    (* signed multiplication; temp is double DEST size*)
            DEST ← DEST ∗ SRC  (* signed multiplication *)
            IF temp ≠ DEST
                THEN CF = 1; OF = 1;
                ELSE CF = 0; OF = 0;
            FI;

        ELSE (* NumberOfOperands = 3 *)
            DEST ← SRC1 ∗ SRC2   (* signed multiplication *)
            temp ←SRC1∗ SRC2     (* signed multiplication; temp is double SRC1 size *)
            IF temp ≠ DEST
                THEN CF = 1; OF = 1;
                ELSE CF = 0; OF = 0;
            FI;
    FI;
FI;
```

Flags Affected

For the one operand form of the instruction, the CF and OF flags are set when significant bits are carried into the upper half of the result and cleared when the result fits exactly in the lower half of the result. For the two- and three-operand forms of the instruction, the CF and OF flags are set when the result must be truncated to fit in the destination operand size and cleared when the result fits exactly in the destination operand size. The SF, ZF, AF, and PF flags are undefined.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.

#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

47.3 IN—Input from Port

Opcode	Instruction	Description
E4 *ib*	IN AL,*imm8*	Input byte from *imm8* I/O port address into AL
E5 *ib*	IN AX,*imm8*	Input byte from *imm8* I/O port address into AX
E5 *ib*	IN EAX,*imm8*	Input byte from *imm8* I/O port address into EAX
EC	IN AL,DX	Input byte from I/O port in DX into AL
ED	IN AX,DX	Input word from I/O port in DX into AX
ED	IN EAX,DX	Input doubleword from I/O port in DX into EAX

Description

Copies the value from the I/O port specified with the second operand (source operand) to the destination operand (first operand). The source operand can be a byte-immediate or the DX register; the destination operand can be register AL, AX, or EAX, depending on the size of the port being accessed (8, 16, or 32 bits, respectively). Using the DX register as a source operand allows I/O port addresses from 0 to 65,535 to be accessed; using a byte immediate allows I/O port addresses 0 to 255 to be accessed.

When accessing an 8-bit I/O port, the opcode determines the port size; when accessing a 16- and 32-bit I/O port, the operand-size attribute determines the port size.

At the machine code level, I/O instructions are shorter when accessing 8-bit I/O ports. Here, the upper eight bits of the port address will be 0.

This instruction is only useful for accessing I/O ports located in the processor’s I/O address space. See “Input/Output”, for more information on accessing I/O ports in the I/O address space.

Operation

```
IF ((PE = 1) AND ((CPL > IOPL) OR (VM = 1)))
    THEN (* Protected mode with CPL > IOPL or virtual-8086 mode *)
        IF (Any I/O Permission Bit for I/O port being accessed = 1)
            THEN (* I/O operation is not allowed *)
                #GP(0);
            ELSE ( * I/O operation is allowed *)
                DEST ← SRC; (* Reads from selected I/O port *)
        FI;
    ELSE (Real Mode or Protected Mode with CPL ≤ IOPL *)
        DEST ← SRC; (* Reads from selected I/O port *)
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the CPL is greater than (has less privilege) the I/O privilege level (IOPL) and any of the corresponding I/O permission bits in TSS for the I/O port being accessed is 1.

Real-Address Mode Exceptions

None.

Virtual-8086 Mode Exceptions

#GP(0)	If any of the I/O permission bits in the TSS for the I/O port being accessed is 1.

47.4 INC—Increment by 1

Opcode	Instruction	Description
FE /0	INC *r/m8*	Increment *r/m* byte by 1
FF /0	INC *r/m16*	Increment *r/m* word by 1
FF /0	INC *r/m32*	Increment *r/m* doubleword by 1
40+ *rw*	INC *r16*	Increment word register by 1
40+ *rd*	INC *r32*	Increment doubleword register by 1

Description

Adds 1 to the destination operand, while preserving the state of the CF flag. The destination operand can be a register or a memory location. This instruction allows a loop counter to be updated without disturbing the CF flag. (Use a ADD instruction with an immediate operand of 1 to perform an increment operation that does updates the CF flag.)

Operation

```
DEST ← DEST +1;
```

Flags Affected

The CF flag is not affected. The OF, SF, ZF, AF, and PF flags are set according to the result.

Protected Mode Exceptions

#GP(0)	If the destination operand is located in a nonwritable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

47.5 INS/INSB/INSW/INSD—Input from Port to String

Opcode	Instruction	Description
6C	INS m8, DX	Input byte from I/O port specified in DX into memory location specified in ES:(E)DI
6D	INS m16, DX	Input word from I/O port specified in DX into memory location specified in ES:(E)DI
6D	INS m32, DX	Input doubleword from I/O port specified in DX into memory location specified in ES:(E)DI
6C	INSB	Input byte from I/O port specified in DX into memory location specified with ES:(E)DI
6D	INSW	Input word from I/O port specified in DX into memory location specified in ES:(E)DI
6D	INSD	Input doubleword from I/O port specified in DX into memory location specified in ES:(E)DI

Description

Copies the data from the I/O port specified with the source operand (second operand) to the destination operand (first operand). The source operand is an I/O port address (from 0 to 65,535) that is read from the DX register. The destination operand is a memory location, the address of which is read from either the ES:EDI or the ES:DI registers (depending on the address-size attribute of the instruction, 32 or 16, respectively). (The ES segment cannot be overridden with a segment override prefix.) The size of the I/O port being accessed (that is, the size of the source and destination operands) is determined by the opcode for an 8-bit I/O port or by the operand-size attribute of the instruction for a 16- or 32-bit I/O port.

At the assembly-code level, two forms of this instruction are allowed: the “explicit-operands” form and the “no-operands” form. The explicit-operands form (specified with the INS mnemonic) allows the source and destination operands to be specified explicitly. Here, the source operand must be “DX,” and the destination operand should be a symbol that indicates the size of the I/O port and the destination address. This explicit-operands form is provided to allow documentation; however, note that the documentation provided by this form can be misleading. That is, the destination operand symbol must specify the correct **type** (size) of the operand (byte, word, or doubleword), but it does not have to specify the correct **location**. The location is always specified by the ES:(E)DI registers, which must be loaded correctly before the INS instruction is executed.

The no-operands form provides “short forms” of the byte, word, and doubleword versions of the INS instructions. Here also DX is assumed by the processor to be the source operand and ES:(E)DI is assumed to be the destination operand. The size of the I/O port is specified with the choice of mnemonic: INSB (byte), INSW (word), or INSD (doubleword).

After the byte, word, or doubleword is transfer from the I/O port to the memory location, the (E)DI register is incremented or decremented automatically according to the setting of the DF flag in the EFLAGS register. (If the DF flag is 0, the (E)DI register is incremented; if the DF flag is 1, the (E)DI register is decremented.) The (E)DI register is incremented or decremented by 1 for byte operations, by 2 for word operations, or by 4 for doubleword operations.

The INS, INSB, INSW, and INSD instructions can be preceded by the REP prefix for block input of ECX bytes, words, or doublewords. See “REP/REPE/REPZ/REPNE /REPNZ—Repeat String Operation Prefix” in this chapter for a description of the REP prefix.

These instructions are only useful for accessing I/O ports located in the processor’s I/O address space. See“Input/Output”, for more information on accessing I/O ports in the I/O address space.

Operation

```
IF ((PE = 1) AND ((CPL > IOPL) OR (VM = 1)))
    THEN (* Protected mode with CPL > IOPL or virtual-8086 mode *)
        IF (Any I/O Permission Bit for I/O port being accessed = 1)
            THEN (* I/O operation is not allowed *)
                #GP(0);
            ELSE ( * I/O operation is allowed *)
                DEST ← SRC; (* Reads from I/O port *)
        FI;
    ELSE (Real Mode or Protected Mode with CPL ≤ IOPL *)
        DEST ← SRC; (* Reads from I/O port *)
FI;
IF (byte transfer)
    THEN IF DF = 0
        THEN (E)DI ← (E)DI + 1;
        ELSE (E)DI ← (E)DI – 1;
    FI;
    ELSE IF (word transfer)
        THEN IF DF = 0
            THEN (E)DI ← (E)DI + 2;
            ELSE (E)DI ← (E)DI – 2;
        FI;
        ELSE (* doubleword transfer *)
            THEN IF DF = 0
```

```
                THEN (E)DI ← (E)DI + 4;
                ELSE (E)DI ← (E)DI – 4;
            FI;
    FI;
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the CPL is greater than (has less privilege) the I/O privilege level (IOPL) and any of the corresponding I/O permission bits in TSS for the I/O port being accessed is 1.
	If the destination is located in a nonwritable segment.
	If an illegal memory operand effective address in the ES segments is given.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.

Virtual-8086 Mode Exceptions

#GP(0)	If any of the I/O permission bits in the TSS for the I/O port being accessed is 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

47.6 INT *n*/INTO/INT 3—Call to Interrupt Procedure

Opcode	Instruction	Description
CC	INT 3	Interrupt 3—trap to debugger
CD *ib*	INT *imm8*	Interrupt vector number specified by immediate byte
CE	INTO	Interrupt 4—if overflow flag is 1

Description

The INT *n* instruction generates a call to the interrupt or exception handler specified with the destination operand (see “Interrupts and Exceptions”). The destination operand specifies an interrupt vector number from 0 to 255, encoded as an 8-bit unsigned intermediate value. Each interrupt vector number provides an index to a gate descriptor in the IDT. The first 32 interrupt vector numbers are reserved by Intel for system use. Some of these interrupts are used for internally generated exceptions.

The INT *n* instruction is the general mnemonic for executing a software-generated call to an interrupt handler. The INTO instruction is a special mnemonic for calling overflow exception (#OF), interrupt vector number 4. The overflow interrupt checks the OF flag in the EFLAGS register and calls the overflow interrupt handler if the OF flag is set to 1.

The INT 3 instruction generates a special one byte opcode (CC) that is intended for calling the debug exception handler. (This one byte form is valuable because it can be used to replace the first byte of any instruction with a breakpoint, including other one byte instructions, without over-writing other code). To further support its function as a debug breakpoint, the interrupt generated with the CC opcode also differs from the regular software interrupts as follows:

- Interrupt redirection does not happen when in VME mode; the interrupt is handled by a protected-mode handler.
- The virtual-8086 mode IOPL checks do not occur. The interrupt is taken without faulting at any IOPL level.

Note that the "normal" 2-byte opcode for INT 3 (CD03) does not have these special features. Intel and Microsoft assemblers will not generate the CD03 opcode from any mnemonic, but this opcode can be created by direct numeric code definition or by self-modifying code.

The action of the INT *n* instruction (including the INTO and INT 3 instructions) is similar to that of a far call made with the CALL instruction. The primary difference is that with the INT *n* instruction, the EFLAGS register is pushed onto the stack before the return address. (The return address is a far address consisting of the current values of the CS and EIP registers.) Returns from interrupt procedures are handled with the IRET instruction, which pops the EFLAGS information and return address from the stack.

The interrupt vector number specifies an interrupt descriptor in the interrupt descriptor table (IDT); that is, it provides index into the IDT. The selected interrupt descriptor in turn contains a pointer to an interrupt or exception handler procedure. In protected mode, the IDT contains an array of 8-byte descriptors, each of which is an interrupt gate, trap gate, or task gate. In real-address mode, the IDT is an array of 4-byte far pointers (2-byte code segment selector and a 2-byte instruction pointer), each of which point directly to a procedure in the selected segment. (Note that in real-address mode, the IDT is called the **interrupt vector table**, and it's pointers are called interrupt vectors.)

The following decision table indicates which action in the lower portion of the table is taken given the conditions in the upper portion of the table. Each Y in the lower section of the decision table represents a procedure defined in the "Operation" section for this instruction (except #GP).

PE	0	1	1	1	1	1	1	1
VM	–	–	–	–	–	0	1	1
IOPL	–	–	–	–	–	–	<3	=3
DPL/CPL RELATIONSHIP	–	DPL< CPL	–	DPL> CPL	DPL= CPL or C	DPL< CPL & NC	–	–
INTERRUPT TYPE	–	S/W	–	–	–	–	–	–
GATE TYPE	–	–	Task	Trap or Interrupt	Trap or Interrupt	Trap or Interrupt	Trap or Interrupt	Trap or Interrupt
REAL-ADDRESS-MODE	Y							
PROTECTED-MODE		Y	Y	Y	Y	Y	Y	Y

TRAP-OR-INTERRUPT-GATE				Y	Y	Y	Y	Y
INTER-PRIVILEGE-LEVEL-INTERRUPT						Y		
INTRA-PRIVILEGE-LEVEL-INTERRUPT					Y			
INTERRUPT-FROM-VIRTUAL-8086-MODE								Y
TASK-GATE			Y					
#GP		Y		Y			Y	

NOTES:
1. –Don’t Care.
2. YYes, Action Taken.
3. BlankAction Not Taken.

When the processor is executing in virtual-8086 mode, the IOPL determines the action of the INT *n* instruction. If the IOPL is less than 3, the processor generates a general protection exception (#GP); if the IOPL is 3, the processor executes a protected mode interrupt to privilege level 0. The interrupt gate’s DPL must be set to three and the target CPL of the interrupt handler procedure must be 0 to execute the protected mode interrupt to privilege level 0.

The interrupt descriptor table register (IDTR) specifies the base linear address and limit of the IDT. The initial base address value of the IDTR after the processor is powered up or reset is 0.

Operation

The following operational description applies not only to the INT *n* and INTO instructions, but also to external interrupts and exceptions.

```
IF PE=0
    THEN
        GOTO REAL-ADDRESS-MODE;
    ELSE (* PE=1 *)
        IF (VM=1 AND IOPL < 3 AND INT n)
            THEN
                #GP(0);
            ELSE (* protected mode or virtual-8086 mode interrupt *)
                GOTO PROTECTED-MODE;
        FI;
FI;

REAL-ADDRESS-MODE:
    IF ((DEST ∗ 4) + 3) is not within IDT limit THEN #GP; FI;
    IF stack not large enough for a 6-byte return information THEN #SS; FI;
    Push (EFLAGS[15:0]);
    IF ← 0; (* Clear interrupt flag *)
    TF ← 0; (* Clear trap flag *)
    AC ← 0; (*Clear AC flag*)
    Push(CS);
    Push(IP);
    (* No error codes are pushed *)
    CS ← IDT(Descriptor (vector_number ∗ 4), selector));
    EIP ← IDT(Descriptor (vector_number ∗ 4), offset)); (* 16 bit offset AND 0000FFFFH *)
END;

PROTECTED-MODE:
    IF ((DEST ∗ 8) + 7) is not within IDT limits
        OR selected IDT descriptor is not an interrupt-, trap-, or task-gate type
            THEN #GP((DEST ∗ 8) + 2 + EXT);
            (* EXT is bit 0 in error code *)
    FI;
    IF software interrupt (* generated by INT n, INT 3, or INTO *)
        THEN
            IF gate descriptor DPL < CPL
                THEN #GP((vector_number ∗ 8) + 2 );
                (* PE=1, DPL<CPL, software interrupt *)
            FI;
        FI;
        IF gate not present THEN #NP((vector_number ∗ 8) + 2 + EXT); FI;
        IF task gate (* specified in the selected interrupt table descriptor *)
            THEN GOTO TASK-GATE;
            ELSE GOTO TRAP-OR-INTERRUPT-GATE; (* PE=1, trap/interrupt gate *)
        FI;
    END;

    TASK-GATE: (* PE=1, task gate *)
        Read segment selector in task gate (IDT descriptor);
            IF local/global bit is set to local
                OR index not within GDT limits
                    THEN #GP(TSS selector);
            FI;
            Access TSS descriptor in GDT;
            IF TSS descriptor specifies that the TSS is busy (low-order 5 bits set to 00001)
                THEN #GP(TSS selector);
            FI;
            IF TSS not present
                THEN #NP(TSS selector);
            FI;
        SWITCH-TASKS (with nesting) to TSS;
        IF interrupt caused by fault with error code
            THEN
                IF stack limit does not allow push of error code
                    THEN #SS(0);
                FI;
                Push(error code);
        FI;
        IF EIP not within code segment limit
            THEN #GP(0);
        FI;
    END;
    TRAP-OR-INTERRUPT-GATE
        Read segment selector for trap or interrupt gate (IDT descriptor);
        IF segment selector for code segment is null
            THEN #GP(0H + EXT); (* null selector with EXT flag set *)
        FI;
        IF segment selector is not within its descriptor table limits
            THEN #GP(selector + EXT);
        FI;
        Read trap or interrupt handler descriptor;
        IF descriptor does not indicate a code segment
            OR code segment descriptor DPL > CPL
                THEN #GP(selector + EXT);
        FI;
        IF trap or interrupt gate segment is not present,
            THEN #NP(selector + EXT);
        FI;
        IF code segment is non-conforming AND DPL < CPL
            THEN IF VM=0
                THEN
                    GOTO INTER-PRIVILEGE-LEVEL-INTERRUPT;
                    (* PE=1, interrupt or trap gate, nonconforming *)
                    (* code segment, DPL<CPL, VM=0 *)
                ELSE (* VM=1 *)
                    IF code segment DPL ≠ 0 THEN #GP(new code segment selector); FI;
                    GOTO INTERRUPT-FROM-VIRTUAL-8086-MODE;
                    (* PE=1, interrupt or trap gate, DPL<CPL, VM=1 *)
            FI;
            ELSE (* PE=1, interrupt or trap gate, DPL ≥ CPL *)
                IF VM=1 THEN #GP(new code segment selector); FI;
                IF code segment is conforming OR code segment DPL = CPL
                    THEN
                        GOTO INTRA-PRIVILEGE-LEVEL-INTERRUPT;
                    ELSE
                        #GP(CodeSegmentSelector + EXT);
                        (* PE=1, interrupt or trap gate, nonconforming *)
                        (* code segment, DPL>CPL *)
                FI;
        FI;
    END;

    INTER-PREVILEGE-LEVEL-INTERRUPT
        (* PE=1, interrupt or trap gate, non-conforming code segment, DPL<CPL *)
        (* Check segment selector and descriptor for stack of new privilege level in current TSS *)
        IF current TSS is 32-bit TSS
            THEN
                TSSstackAddress ← (new code segment DPL ∗ 8) + 4
                IF (TSSstackAddress + 7) > TSS limit
                    THEN #TS(current TSS selector); FI;
                NewSS ← TSSstackAddress + 4;
                NewESP ← stack address;
            ELSE (* TSS is 16-bit *)
                TSSstackAddress ← (new code segment DPL ∗ 4) + 2
                IF (TSSstackAddress + 4) > TSS limit
                THEN #TS(current TSS selector); FI;
            NewESP ← TSSstackAddress;
            NewSS ← TSSstackAddress + 2;
    FI;
    IF segment selector is null THEN #TS(EXT); FI;
    IF segment selector index is not within its descriptor table limits
        OR segment selector’s RPL ≠ DPL of code segment,
            THEN #TS(SS selector + EXT);
    FI;
Read segment descriptor for stack segment in GDT or LDT;
    IF stack segment DPL ≠ DPL of code segment,
        OR stack segment does not indicate writable data segment,
            THEN #TS(SS selector + EXT);
    FI;
    IF stack segment not present THEN #SS(SS selector+EXT); FI;
    IF 32-bit gate
        THEN
            IF new stack does not have room for 24 bytes (error code pushed)
                OR 20 bytes (no error code pushed)
                    THEN #SS(segment selector + EXT);
            FI;
        ELSE (* 16-bit gate *)
            IF new stack does not have room for 12 bytes (error code pushed)
                OR 10 bytes (no error code pushed);
                    THEN #SS(segment selector + EXT);
            FI;
    FI;
    IF instruction pointer is not within code segment limits THEN #GP(0); FI;
    SS:ESP ← TSS(NewSS:NewESP) (* segment descriptor information also loaded *)
    IF 32-bit gate
        THEN
            CS:EIP ← Gate(CS:EIP); (* segment descriptor information also loaded *)
        ELSE (* 16-bit gate *)
            CS:IP ← Gate(CS:IP); (* segment descriptor information also loaded *)
    FI;
    IF 32-bit gate
        THEN
            Push(far pointer to old stack); (* old SS and ESP, 3 words padded to 4 *);
            Push(EFLAGS);
            Push(far pointer to return instruction); (* old CS and EIP, 3 words padded to 4*);
            Push(ErrorCode); (* if needed, 4 bytes *)
        ELSE(* 16-bit gate *)
            Push(far pointer to old stack); (* old SS and SP, 2 words *);
            Push(EFLAGS(15..0));
            Push(far pointer to return instruction); (* old CS and IP, 2 words *);
            Push(ErrorCode); (* if needed, 2 bytes *)
    FI;
    CPL ← CodeSegmentDescriptor(DPL);
    CS(RPL) ← CPL;
    IF interrupt gate
        THEN IF ← 0 (* interrupt flag to 0 (disabled) *); FI;
    TF ← 0;
    VM ← 0;
    RF ← 0;
    NT ← 0;
    END;

INTERRUPT-FROM-VIRTUAL-8086-MODE:
    (* Check segment selector and descriptor for privilege level 0 stack in current TSS *)
    IF current TSS is 32-bit TSS
        THEN
            TSSstackAddress ← (new code segment DPL ∗ 8) + 4
            IF (TSSstackAddress + 7) > TSS limit
                THEN #TS(current TSS selector); FI;
            NewSS ← TSSstackAddress + 4;
            NewESP ← stack address;
        ELSE (* TSS is 16-bit *)
            TSSstackAddress ← (new code segment DPL ∗ 4) + 2
            IF (TSSstackAddress + 4) > TSS limit
                THEN #TS(current TSS selector); FI;
            NewESP ← TSSstackAddress;
            NewSS ← TSSstackAddress + 2;
    FI;
        IF segment selector is null THEN #TS(EXT); FI;
        IF segment selector index is not within its descriptor table limits
            OR segment selector’s RPL ≠ DPL of code segment,
                THEN #TS(SS selector + EXT);
        FI;
    Access segment descriptor for stack segment in GDT or LDT;
    IF stack segment DPL ≠ DPL of code segment,
        OR stack segment does not indicate writable data segment,
            THEN #TS(SS selector + EXT);
    FI;
    IF stack segment not present THEN #SS(SS selector+EXT); FI;
    IF 32-bit gate
        THEN
```

```
                IF new stack does not have room for 40 bytes (error code pushed)
                    OR 36 bytes (no error code pushed);
                        THEN #SS(segment selector + EXT);
                FI;
            ELSE (* 16-bit gate *)
                IF new stack does not have room for 20 bytes (error code pushed)
                    OR 18 bytes (no error code pushed);
                        THEN #SS(segment selector + EXT);
                FI;
        FI;
        IF instruction pointer is not within code segment limits THEN #GP(0); FI;
        tempEFLAGS ← EFLAGS;
        VM ← 0;
        TF ← 0;
        RF ← 0;
        IF service through interrupt gate THEN IF ← 0; FI;
        TempSS ← SS;
        TempESP ← ESP;
        SS:ESP ← TSS(SS0:ESP0); (* Change to level 0 stack segment *)
        (* Following pushes are 16 bits for 16-bit gate and 32 bits for 32-bit gates *)
        (* Segment selector pushes in 32-bit mode are padded to two words *)
        Push(GS);
        Push(FS);
        Push(DS);
        Push(ES);
        Push(TempSS);
        Push(TempESP);
        Push(TempEFlags);
        Push(CS);
        Push(EIP);
        GS ← 0; (*segment registers nullified, invalid in protected mode *)
        FS ← 0;
        DS ← 0;
        ES ← 0;
        CS ← Gate(CS);
        IF OperandSize=32
            THEN
                EIP ← Gate(instruction pointer);
            ELSE (* OperandSize is 16 *)
                EIP ← Gate(instruction pointer) AND 0000FFFFH;
        FI;
        (* Starts execution of new routine in Protected Mode *)
    END;
    INTRA-PRIVILEGE-LEVEL-INTERRUPT:
        (* PE=1, DPL = CPL or conforming segment *)
        IF 32-bit gate
            THEN
                IF current stack does not have room for 16 bytes (error code pushed)
                    OR 12 bytes (no error code pushed); THEN #SS(0);
                FI;
            ELSE (* 16-bit gate *)
                IF current stack does not have room for 8 bytes (error code pushed)
                    OR 6 bytes (no error code pushed); THEN #SS(0);
                FI;
        IF instruction pointer not within code segment limit THEN #GP(0); FI;
        IF 32-bit gate
            THEN
                Push (EFLAGS);
                Push (far pointer to return instruction); (* 3 words padded to 4 *)
                CS:EIP ← Gate(CS:EIP); (* segment descriptor information also loaded *)
                Push (ErrorCode); (* if any *)
            ELSE (* 16-bit gate *)
                Push (FLAGS);
                Push (far pointer to return location); (* 2 words *)
                CS:IP ← Gate(CS:IP); (* segment descriptor information also loaded *)
                Push (ErrorCode); (* if any *)
        FI;
        CS(RPL) ← CPL;
        IF interrupt gate
            THEN
                IF ← 0; FI;
                TF ← 0;
                NT ← 0;
                VM ← 0;
                RF ← 0;
        FI;
    END;
```

Flags Affected

The EFLAGS register is pushed onto the stack. The IF, TF, NT, AC, RF, and VM flags may be cleared, depending on the mode of operation of the processor when the INT instruction is executed (see the "Operation" section). If the interrupt uses a task gate, any flags may be set or cleared, controlled by the EFLAGS image in the new task's TSS.

Protected Mode Exceptions

#GP(0)	If the instruction pointer in the IDT or in the interrupt-, trap-, or task gate is beyond the code segment limits.
#GP(selector)	If the segment selector in the interrupt-, trap-, or task gate is null.
	If a interrupt-, trap-, or task gate, code segment, or TSS segment selector index is outside its descriptor table limits.
	If the interrupt vector number is outside the IDT limits.
	If an IDT descriptor is not an interrupt-, trap-, or task-descriptor.
	If an interrupt is generated by the INT *n*, INT 3, or INTO instruction and the DPL of an interrupt-, trap-, or task-descriptor is less than the CPL.
	If the segment selector in an interrupt- or trap-gate does not point to a segment descriptor for a code segment.
	If the segment selector for a TSS has its local/global bit set for local.
	If a TSS segment descriptor specifies that the TSS is busy or not available.
#SS(0)	If pushing the return address, flags, or error code onto the stack exceeds the bounds of the stack segment and no stack switch occurs.
#SS(selector)	If the SS register is being loaded and the segment pointed to is marked not present.
	If pushing the return address, flags, error code, or stack segment pointer exceeds the bounds of the new stack segment when a stack switch occurs.
#NP(selector)	If code segment, interrupt-, trap-, or task gate, or TSS is not present.
#TS(selector)	If the RPL of the stack segment selector in the TSS is not equal to the DPL of the code segment being accessed by the interrupt or trap gate.
	If DPL of the stack segment descriptor pointed to by the stack segment selector in the TSS is not equal to the DPL of the code segment descriptor for the interrupt or trap gate.
	If the stack segment selector in the TSS is null.
	If the stack segment for the TSS is not a writable data segment.
	If segment-selector index for stack segment is outside descriptor table limits.
#PF(fault-code)	If a page fault occurs.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

	If the interrupt vector number is outside the IDT limits.
#SS	If stack limit violation on push.
	If pushing the return address, flags, or error code onto the stack exceeds the bounds of the stack segment.

Virtual-8086 Mode Exceptions

#GP(0)	(For INT *n*, INTO, or BOUND instruction) If the IOPL is less than 3 or the DPL of the interrupt-, trap-, or task-gate descriptor is not equal to 3.
	If the instruction pointer in the IDT or in the interrupt-, trap-, or task gate is beyond the code segment limits.
#GP(selector)	If the segment selector in the interrupt-, trap-, or task gate is null.
	If a interrupt-, trap-, or task gate, code segment, or TSS segment selector index is outside its descriptor table limits.
	If the interrupt vector number is outside the IDT limits.
	If an IDT descriptor is not an interrupt-, trap-, or task-descriptor.
	If an interrupt is generated by the INT *n* instruction and the DPL of an interrupt-, trap-, or task-descriptor is less than the CPL.
	If the segment selector in an interrupt- or trap-gate does not point to a segment descriptor for a code segment.
	If the segment selector for a TSS has its local/global bit set for local.
#SS(selector)	If the SS register is being loaded and the segment pointed to is marked not present.
	If pushing the return address, flags, error code, stack segment pointer, or data segments exceeds the bounds of the stack segment.
#NP(selector)	If code segment, interrupt-, trap-, or task gate, or TSS is not present.
#TS(selector)	If the RPL of the stack segment selector in the TSS is not equal to the DPL of the code segment being accessed by the interrupt or trap gate.
	If DPL of the stack segment descriptor for the TSS's stack segment is not equal to the DPL of the code segment descriptor for the interrupt or trap gate.
	If the stack segment selector in the TSS is null.
	If the stack segment for the TSS is not a writable data segment.
	If segment-selector index for stack segment is outside descriptor table limits.
#PF(fault-code)	If a page fault occurs.
#BP	If the INT 3 instruction is executed.
#OF	If the INTO instruction is executed and the OF flag is set.

47.7 INVD—Invalidate Internal Caches

Opcode	Instruction	Description
0F 08	INVD	Flush internal caches; initiate flushing of external caches.

Description

Invalidates (flushes) the processor’s internal caches and issues a special-function bus cycle that directs external caches to also flush themselves. Data held in internal caches is not written back to main memory.

After executing this instruction, the processor does not wait for the external caches to complete their flushing operation before proceeding with instruction execution. It is the responsibility of hardware to respond to the cache flush signal.

The INVD instruction is a privileged instruction. When the processor is running in protected mode, the CPL of a program or procedure must be 0 to execute this instruction.

Use this instruction with care. Data cached internally and not written back to main memory will be lost. Unless there is a specific requirement or benefit to flushing caches without writing back modified cache lines (for example, testing or fault recovery where cache coherency with main memory is not a concern), software should use the WBINVD instruction.

Intel Architecture Compatibility

The INVD instruction is implementation dependent, and its function may be implemented differently on future Intel Architecture processors. This instruction is not supported on Intel Architecture processors earlier than the Intel486 processor.

Operation

```
Flush(InternalCaches);
SignalFlush(ExternalCaches);
Continue (* Continue execution);
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0.

Real-Address Mode Exceptions

None.

Virtual-8086 Mode Exceptions

#GP(0)	The INVD instruction cannot be executed in virtual-8086 mode.

47.8 INVLPG—Invalidate TLB Entry

Opcode	Instruction	Description
0F 01/7	INVLPG m	Invalidate TLB Entry for page that contains *m*

Description

Invalidates (flushes) the translation lookaside buffer (TLB) entry specified with the source operand. The source operand is a memory address. The processor determines the page that contains that address and flushes the TLB entry for that page.

The INVLPG instruction is a privileged instruction. When the processor is running in protected mode, the CPL of a program or procedure must be 0 to execute this instruction.

The INVLPG instruction normally flushes the TLB entry only for the specified page; however, in some cases, it flushes the entire TLB. See “MOV—Move to/from Control Registers” in this chapter for further information on operations that flush the TLB.

Intel Architecture Compatibility

The INVLPG instruction is implementation dependent, and its function may be implemented differently on future Intel Architecture processors. This instruction is not supported on Intel Architecture processors earlier than the Intel486 processor.

Operation

```
Flush(RelevantTLBEntries);
Continue (* Continue execution);
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0.
#UD	Operand is a register.

Real-Address Mode Exceptions

#UD	Operand is a register.

Virtual-8086 Mode Exceptions

#GP(0)	The INVLPG instruction cannot be executed at the virtual-8086 mode.

47.9 IRET/IRETD—Interrupt Return

Opcode	Instruction	Description
CF	IRET	Interrupt return (16-bit operand size)
CF	IRETD	Interrupt return (32-bit operand size)

Description

Returns program control from an exception or interrupt handler to a program or procedure that was interrupted by an exception, an external interrupt, or a software-generated interrupt. These instructions are also used to perform a return from a nested task. (A nested task is created when a CALL instruction is used to initiate a task switch or when an interrupt or exception causes a task switch to an interrupt or exception handler.)

IRET and IRETD are mnemonics for the same opcode. The IRETD mnemonic (interrupt return double) is intended for use when returning from an interrupt when using the 32-bit operand size; however, most assemblers use the IRET mnemonic interchangeably for both operand sizes.

In Real-Address Mode, the IRET instruction preforms a far return to the interrupted program or procedure. During this operation, the processor pops the return instruction pointer, return code segment selector, and EFLAGS image from the stack to the EIP, CS, and EFLAGS registers, respectively, and then resumes execution of the interrupted program or procedure.

In Protected Mode, the action of the IRET instruction depends on the settings of the NT (nested task) and VM flags in the EFLAGS register and the VM flag in the EFLAGS image stored on the current stack. Depending on the setting of these flags, the processor performs the following types of interrupt returns:

- Return from virtual-8086 mode.
- Return to virtual-8086 mode.
- Intra-privilege level return.
- Inter-privilege level return.
- Return from nested task (task switch).

If the NT flag (EFLAGS register) is cleared, the IRET instruction performs a far return from the interrupt procedure, without a task switch. The code segment being returned to must be equally or less privileged than the interrupt handler routine (as indicated by the RPL field of the code segment selector popped from the stack). As with a real-address mode interrupt return, the IRET instruction pops the return instruction pointer, return code segment selector, and EFLAGS image from the stack to the EIP, CS, and EFLAGS registers, respectively, and then resumes execution of the interrupted program or procedure. If the return is to another privilege level, the IRET instruction also pops the stack pointer and SS from the stack, before resuming program execution. If the return is to virtual-8086 mode, the processor also pops the data segment registers from the stack.

If the NT flag is set, the IRET instruction performs a task switch (return) from a nested task (a task called with a CALL instruction, an interrupt, or an exception) back to the calling or interrupted task. The updated state of the task executing the IRET instruction is saved in its TSS. If the task is reentered later, the code that follows the IRET instruction is executed.

Operation

```
IF PE = 0
    THEN
        GOTO REAL-ADDRESS-MODE:;
    ELSE
        GOTO PROTECTED-MODE;
FI;

REAL-ADDRESS-MODE;
    IF OperandSize = 32
        THEN
            IF top 12 bytes of stack not within stack limits THEN #SS; FI;
            IF instruction pointer not within code segment limits THEN #GP(0); FI;
            EIP ← Pop();
            CS ← Pop(); (* 32-bit pop, high-order 16-bits discarded *)
            tempEFLAGS ← Pop();
            EFLAGS ← (tempEFLAGS AND 257FD5H) OR (EFLAGS AND 1A0000H);
        ELSE (* OperandSize = 16 *)
            IF top 6 bytes of stack are not within stack limits THEN #SS; FI;
            IF instruction pointer not within code segment limits THEN #GP(0); FI;
            EIP ← Pop();
            EIP ← EIP AND 0000FFFFH;
            CS ← Pop(); (* 16-bit pop *)
            EFLAGS[15:0] ← Pop();
    FI;
END;

PROTECTED-MODE:
    IF VM = 1 (* Virtual-8086 mode: PE=1, VM=1 *)
        THEN
            GOTO RETURN-FROM-VIRTUAL-8086-MODE; (* PE=1, VM=1 *)
    FI;
    IF NT = 1
        THEN
            GOTO TASK-RETURN;( *PE=1, VM=0, NT=1 *)
    FI;
    IF OperandSize=32
        THEN
            IF top 12 bytes of stack not within stack limits
                THEN #SS(0)
            FI;
            tempEIP ← Pop();
            tempCS ← Pop();
            tempEFLAGS ← Pop();
        ELSE (* OperandSize = 16 *)
            IF top 6 bytes of stack are not within stack limits
                THEN #SS(0);
            FI;
            tempEIP ← Pop();
            tempCS ← Pop();
            tempEFLAGS ← Pop();
            tempEIP ← tempEIP AND FFFFH;
            tempEFLAGS ← tempEFLAGS AND FFFFH;
    FI;
    IF tempEFLAGS(VM) = 1 AND CPL=0
        THEN
            GOTO RETURN-TO-VIRTUAL-8086-MODE;
            (* PE=1, VM=1 in EFLAGS image *)
        ELSE
            GOTO PROTECTED-MODE-RETURN;
            (* PE=1, VM=0 in EFLAGS image *)
    FI;

RETURN-FROM-VIRTUAL-8086-MODE:
(* Processor is in virtual-8086 mode when IRET is executed and stays in virtual-8086 mode *)
    IF IOPL=3 (* Virtual mode: PE=1, VM=1, IOPL=3 *)
        THEN IF OperandSize = 32
            THEN
                IF top 12 bytes of stack not within stack limits THEN #SS(0); FI;
                IF instruction pointer not within code segment limits THEN #GP(0); FI;
                EIP ← Pop();
                CS ← Pop(); (* 32-bit pop, high-order 16-bits discarded *)
                EFLAGS ← Pop();
                (*VM,IOPL,VIP,and VIF EFLAGS bits are not modified by pop *)
            ELSE (* OperandSize = 16 *)
                IF top 6 bytes of stack are not within stack limits THEN #SS(0); FI;
                IF instruction pointer not within code segment limits THEN #GP(0); FI;
                EIP ← Pop();
                EIP ← EIP AND 0000FFFFH;
                CS ← Pop(); (* 16-bit pop *)
                EFLAGS[15:0] ← Pop(); (* IOPL in EFLAGS is not modified by pop *)
            FI;
        ELSE
            #GP(0); (* trap to virtual-8086 monitor: PE=1, VM=1, IOPL<3 *)
    FI;
END;

RETURN-TO-VIRTUAL-8086-MODE:
(* Interrupted procedure was in virtual-8086 mode: PE=1, VM=1 in flags image *)
    IF top 24 bytes of stack are not within stack segment limits
        THEN #SS(0);
    FI;
    IF instruction pointer not within code segment limits
        THEN #GP(0);
    FI;
    CS ← tempCS;
    EIP ← tempEIP;
    EFLAGS ← tempEFLAGS
    TempESP ← Pop();
    TempSS ← Pop();
    ES ← Pop(); (* pop 2 words; throw away high-order word *)
    DS ← Pop(); (* pop 2 words; throw away high-order word *)
    FS ← Pop(); (* pop 2 words; throw away high-order word *)
    GS ← Pop(); (* pop 2 words; throw away high-order word *)
    SS:ESP ← TempSS:TempESP;
    (* Resume execution in Virtual-8086 mode *)
END;

TASK-RETURN: (* PE=1, VM=1, NT=1 *)
    Read segment selector in link field of current TSS;
    IF local/global bit is set to local
        OR index not within GDT limits
            THEN #GP(TSS selector);
    FI;
    Access TSS for task specified in link field of current TSS;
    IF TSS descriptor type is not TSS or if the TSS is marked not busy
        THEN #GP(TSS selector);
    FI;
    IF TSS not present
        THEN #NP(TSS selector);
    FI;
    SWITCH-TASKS (without nesting) to TSS specified in link field of current TSS;
    Mark the task just abandoned as NOT BUSY;
    IF EIP is not within code segment limit
        THEN #GP(0);
    FI;
END;

PROTECTED-MODE-RETURN: (* PE=1, VM=0 in flags image *)
    IF return code segment selector is null THEN GP(0); FI;
    IF return code segment selector addrsses descriptor beyond descriptor table limit
        THEN GP(selector; FI;
    Read segment descriptor pointed to by the return code segment selector
    IF return code segment descriptor is not a code segment THEN #GP(selector); FI;
    IF return code segment selector RPL < CPL THEN #GP(selector); FI;
    IF return code segment descriptor is conforming
        AND return code segment DPL > return code segment selector RPL
            THEN #GP(selector); FI;
    IF return code segment descriptor is not present THEN #NP(selector); FI:
    IF return code segment selector RPL > CPL
        THEN GOTO RETURN-OUTER-PRIVILEGE-LEVEL;
        ELSE GOTO RETURN-TO-SAME-PRIVILEGE-LEVEL
    FI;
END;

RETURN-TO-SAME-PRIVILEGE-LEVEL: (* PE=1, VM=0 in flags image, RPL=CPL *)
    IF EIP is not within code segment limits THEN #GP(0); FI;
    EIP ← tempEIP;
    CS ← tempCS; (* segment descriptor information also loaded *)
    EFLAGS (CF, PF, AF, ZF, SF, TF, DF, OF, NT) ← tempEFLAGS;
    IF OperandSize=32
        THEN
            EFLAGS(RF, AC, ID) ← tempEFLAGS;
    FI;
    IF CPL ≤ IOPL
        THEN
            EFLAGS(IF) ← tempEFLAGS;
    FI;
    IF CPL = 0
        THEN
            EFLAGS(IOPL) ← tempEFLAGS;
            IF OperandSize=32
                THEN EFLAGS(VM, VIF, VIP) ← tempEFLAGS;
            FI;
    FI;
END;

RETURN-TO-OUTER-PRIVILGE-LEVEL:
    IF OperandSize=32
        THEN
```

```
            IF top 8 bytes on stack are not within limits THEN #SS(0); FI;
        ELSE (* OperandSize=16 *)
            IF top 4 bytes on stack are not within limits THEN #SS(0); FI;
    FI;
    Read return segment selector;
    IF stack segment selector is null THEN #GP(0); FI;
    IF return stack segment selector index is not within its descriptor table limits
            THEN #GP(SSselector); FI;
    Read segment descriptor pointed to by return segment selector;
    IF stack segment selector RPL ≠ RPL of the return code segment selector
        IF stack segment selector RPL ≠ RPL of the return code segment selector
        OR the stack segment descriptor does not indicate a a writable data segment;
        OR stack segment DPL ≠ RPL of the return code segment selector
                THEN #GP(SS selector);
        FI;
        IF stack segment is not present THEN #SS(SS selector); FI;
    IF tempEIP is not within code segment limit THEN #GP(0); FI;
    EIP ← tempEIP;
    CS ← tempCS;
    EFLAGS (CF, PF, AF, ZF, SF, TF, DF, OF, NT) ← tempEFLAGS;
    IF OperandSize=32
        THEN
            EFLAGS(RF, AC, ID) ← tempEFLAGS;
    FI;
    IF CPL ≤ IOPL
        THEN
            EFLAGS(IF) ← tempEFLAGS;
    FI;
    IF CPL = 0
        THEN
            EFLAGS(IOPL) ← tempEFLAGS;
            IF OperandSize=32
                THEN EFLAGS(VM, VIF, VIP) ← tempEFLAGS;
            FI;
    FI;
    CPL ← RPL of the return code segment selector;
    FOR each of segment register (ES, FS, GS, and DS)
        DO;
            IF segment register points to data or non-conforming code segment
            AND CPL > segment descriptor DPL (* stored in hidden part of segment register *)
                THEN (* segment register invalid *)
                    SegmentSelector ← 0; (* null segment selector *)
            FI;
        OD;
END:
```

Flags Affected

All the flags and fields in the EFLAGS register are potentially modified, depending on the mode of operation of the processor. If performing a return from a nested task to a previous task, the EFLAGS register will be modified according to the EFLAGS image stored in the previous task's TSS.

Protected Mode Exceptions

#GP(0)	If the return code or stack segment selector is null.
	If the return instruction pointer is not within the return code segment limit.
#GP(selector)	If a segment selector index is outside its descriptor table limits.
	If the return code segment selector RPL is greater than the CPL.
	If the DPL of a conforming-code segment is greater than the return code segment selector RPL.
	If the DPL for a nonconforming-code segment is not equal to the RPL of the code segment selector.
	If the stack segment descriptor DPL is not equal to the RPL of the return code segment selector.
	If the stack segment is not a writable data segment.

	If the stack segment selector RPL is not equal to the RPL of the return code segment selector.
	If the segment descriptor for a code segment does not indicate it is a code segment.
	If the segment selector for a TSS has its local/global bit set for local.
	If a TSS segment descriptor specifies that the TSS is busy or not available.
#SS(0)	If the top bytes of stack are not within stack limits.
#NP(selector)	If the return code or stack segment is not present.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference occurs when the CPL is 3 and alignment checking is enabled.

Real-Address Mode Exceptions

#GP	If the return instruction pointer is not within the return code segment limit.
#SS	If the top bytes of stack are not within stack limits.

Virtual-8086 Mode Exceptions

#GP(0)	If the return instruction pointer is not within the return code segment limit.
	IF IOPL not equal to 3
#PF(fault-code)	If a page fault occurs.
#SS(0)	If the top bytes of stack are not within stack limits.
#AC(0)	If an unaligned memory reference occurs and alignment checking is enabled.